A Formal Soundness Proof of Region-based Memory Management for Object-Oriented Paradigm *

Florin Craciun[1], Shengchao Qin[1], and Wei-Ngan Chin[2]

[1] Department of Computer Science, Durham University, UK
{florin.craciun,shengchao.qin}@durham.ac.uk
[2] Department of Computer Science, National University of Singapore, Singapore
chinwn@comp.nus.edu.sg

Abstract. Region-based memory management has been proposed as a viable alternative to garbage collection for real-time applications and embedded software. In our previous work we have developed a region type inference algorithm that provides an automatic compile-time region-based memory management for object-oriented paradigm. In this work we present a formal soundness proof of the region type system that is the target of our region inference. More precisely, we prove that the object-oriented programs accepted by our region type system achieve region-based memory management in a safe way. That means, the regions follow a stack-of-regions discipline and regions deallocation never create dangling references in the store and on the program stack. Our contribution is to provide a simple syntactic proof that is based on induction and follows the standard steps of a type safety proof. In contrast the previous safety proofs provided for other region type systems employ quite elaborate techniques.

1 Introduction

Modern object-oriented programming languages provide a run-time system that automatically reclaims memory using tracing garbage collection [24]. A correct garbage collector can guarantee that the memory is not collecting too early, and also that all memory is eventually reclaimed if the program terminates. However the space and time requirements of garbage-collected programs are very difficult to estimate in practice. Therefore many different solutions have been proposed for real-time applications and embedded software running on resource-limited platforms. These solutions either completely omit the use of garbage collectors (e.g. JavaCard platform), or use real-time garbage collectors [1], or use region-based memory management (e.g. Real-Time Specification for Java (RTSJ) [3]).

Region-based memory management systems allocate each new object into a program-specified *region*, with the entire set of objects in each region deallocated simultaneously when the region is deleted. Various studies have shown that region-based memory management can provide memory management with good real-time performance. Individual object deallocation is accurate but time unpredictable, while region deletion presents a better temporal behavior, at the cost of some space overhead. Data locality may also

* The work is supported in part by the EPSRC project EP/E021948/1.

improve when related objects are placed together in the same region. Classifying objects into regions based on their lifetimes may deliver better memory utilization if regions are deleted in a timely manner.

The first safe region-based memory system was introduced by Tofte and Talpin [22, 23] for a functional language. Using a region type inference system, they have provided an automatic static region-based memory management for Standard ML. More precisely, their compiler can group heap allocations into regions and it can statically determine the program points where it is safe to deallocate the regions. Later, several projects have investigated the use of region-based memory management for C-like languages (e.g. Cyclone [13]) and object-oriented languages [9, 5]. These projects provide region type checkers and require programmers to annotate their programs with region declarations. The type checkers then use these declarations to verify that well-typed programs safely use the region-based memory.

In our previous work [8], we have developed the first automatic region type inference system for object-oriented paradigm. Our compiler automatically augments unannotated object-oriented programs with regions type declarations and inserts region allocation/deallocation instructions that achieve a safe memory management. In this paper we provide the safety proof of our region type system that is the target of our previous region inference algorithm.

In our work, we use *lexically-scoped regions* such that the memory is organised as a *stack of regions*, as illustrated in Fig. 1. Regions are memory blocks that are allocated and deallocated by the construct `letreg r in e`, where the region `r` can only be used to allocate objects in the program `e`. The older regions (with longer lifetime) are allocated at the bottom of the stack while the younger regions (with shorter lifetime) are at the top. The region lifetime relations are expressed using a transitive *outlive relation*, denoted by $\succeq$. Thus, we can define the lifetime constraints $\texttt{r0}\succeq\texttt{r1}\wedge\texttt{r1}\succeq\texttt{r2}\wedge\texttt{r2}\succeq\texttt{r3}\wedge\texttt{r3}\succeq\texttt{r4}$ on the regions of Fig. 1. Region lifetime constraints (as shown in Fig. 2) are of two main forms $r_1\succeq r_2$ and $r_1{=}r_2$. The constraint $r_1\succeq r_2$ indicates that the lifetime of region r_1 is not shorter than that of r_2, while the constraint $r_1{=}r_2$ denotes that r_1 and r_2 must be the same region. The equality can be expressed as an outlive relation such that $r_1{=}r_2$ iff $r_1\succeq r_2$ and $r_2\succeq r_1$.

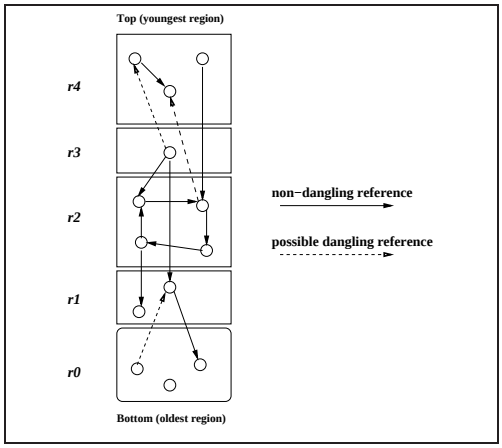

Fig. 1. Lexically-Scoped Regions

Dangling references are a safety issue for region-based memory management. Fig. 1 shows two kinds of references: non-dangling references and possible dangling references. Non-dangling references originate from objects placed in a younger region and point to objects placed either in an older region or inside the same region. Possible dangling references occur when objects placed in an older region point to objects placed in

a younger region. They turn into dangling references when the younger region is deallocated. Using a dangling reference to access memory is unsafe because the accessed memory may have been recycled to store other objects. There are two approaches to eliminating this problem. The first approach allows the program to create dangling references, but uses an effect-based region type system to ensure that the program never accesses memory through a dangling reference [22, 23, 9, 13]. The second approach uses a region type system to prevent the program from creating dangling references at all [5]. Our work has adopted the second approach.

Contributions. The main contribution of this paper is the soundness proof of our region type system for object-oriented paradigm. We prove that our region type system guarantees that well-typed programs use lexically-scoped regions and never create dangling references in the store and on the program stack. We provide a simple syntactic proof based on induction (rather than a more elaborate co-induction machinery), that follows the standard steps of a type safety proof [25]. Our small-step dynamic semantics decomposes high-level expression `letreg r in e` into three intermediate operations: allocation of region `r` on the stack, evaluation of program `e`, and deallocation of region `r`. The difficulty is to prove that after deallocation of region `r`, the store, the program stack and the remaining code do not contain any reference to region `r` and to the objects stored in region `r`. To prove that region deallocation is safe, we use the region constraints of our type system and a syntactic condition that we imposed to restrict the valid intermediate code. However our syntactic restriction does not restrict high-level source code, it only defines the correct intermediate code to which high-level code can be evaluated.

Related Work. In the original effect-based region type system, Tofte and Talpin [23, 21, 2] and later Christiansen and Velschow [9], in their region calculus for object-oriented languages make use of co-induction to prove the soundness. Their proof requires co-induction partly because they prove two properties at the same time: type soundness and translation soundness. The latter property guarantees that there exists a semantic relation between source program and its region-annotated counterpart. Our safety theorems are only focused on the problem of type soundness, thus are simpler to prove. A co-inductive definition is required in their proof also because they use a big-step semantics where certain information is lost when deleting a region from the store, as discussed in [15, 7]. Our system uses a small-step operational semantics instrumented with regions which makes the consistency definition and the proof easier. Calcagno [6] uses a stratified operational semantics to avoid co-induction in the proof of safety properties of a simple version of Tofte and Talpin's region calculus, while Helsen et al. [15, 14] introduces a special constant for defunct regions in their big-step semantics which makes the soundness proof simpler. A similar proof with ours is the safety proof of Niss [19], that in addition to a simple functional language handles an imperative calculus, and like our proof avoids explicit co-induction by using store typing. Cyclone [13] also has an effect system used for a soundness proof and does not use co-induction. Elsman [12] refines Tofte and Talpin's region type system in order to forbid the dangling references and proves by induction the safety for a small functional language. There are many differences between his proof and ours. His proof is based on a small-step contextual semantics [17], while in our proof we explicitly model the heap as a stack of

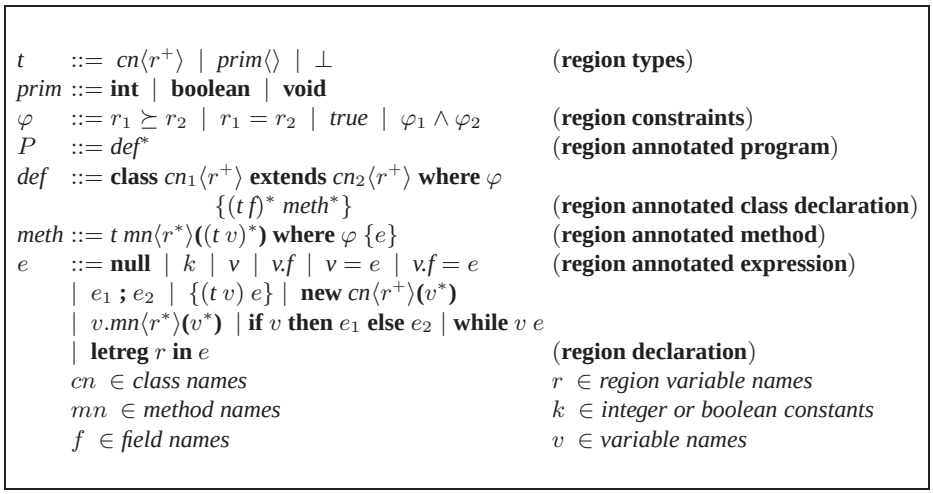

Fig. 2. The Syntax of Region-Annotated Core-Java

regions and we use a consistency relation between the static and dynamic semantics. In addition Elsman uses a syntax-directed containment relation to express the regions of the program values and also to force the stack discipline for regions' allocation and deallocation. In our case the region requirements and the order among regions are expressed by the region constraints of the type system. However we also impose a syntactic condition to restrict the valid intermediate (non-source) programs. Boudol [4] refines Tofte and Talpin's region calculus to a flow-sensitive effect-based region type system, that explicitly records the deallocations effects. He provides a simple proof for a functional language by means of a subject reduction property up to simulation. Although his simulation is half-bisimulation, his proof does not employ co-induction. In contrast our region type system is a flow-insensitive calculus. However our syntactic restriction on intermediate code has a similar role as the flow-sensitive deallocation effect. Our type system is similar to SafeJava's type system of Boyapati et al. [5], but in addition we support the region subtyping principle [13]. However SafeJava does not provide a formal proof for its region type system.

Outline. The paper is organized as follows. Section 2 introduces the syntax of our region calculus. Section 3 presents our region type system, while Section 4 defines the dynamic semantics of our region calculus. Section 5 extends the static semantics to intermediate expressions, while Section 6 presents the soundness theorems. A brief conclusion is given. The technical report [11] contains the details of our inductive proofs.

2 Region Calculus

Our region calculus is designed by annotating with regions a Java-like object-oriented language, named Core-Java [10]. The full syntax of the region-annotated Core-Java language is given in Fig. 2. Core-Java is designed in the same minimalist spirit as the pure functional calculus Featherweight Java [16]. Despite its expression-oriented syntax, Core-Java supports imperative features.

Each class definition is parameterized with one or more regions to form a *region type*. For instance, a region type $cn\langle r_1, ..., r_n\rangle$ is a class name cn annotated with region parameters $r_1...r_n$. Parameterization allows us to obtain a region-polymorphic type for each class whose fields can be allocated in different regions. The first region parameter r_1 is special: it refers to the region in which the instance object of this class is allocated. The fields of the objects, if any, are allocated in the other regions $r_2...r_n$ which should *outlive* the region of the object. This is expressed by the constraint $\bigwedge_{i=2}^{n}(r_i \succeq r_1)$, which captures the property that the regions of the fields (in $r_2...r_n$) should have lifetimes no shorter than the lifetime of the region (namely r_1) of the object that refers to them. This condition, called *no-dangling requirement*, prevents dangling references completely, as it guarantees that each object never references another object in a younger region. In general the class invariant, φ, of a class consists of the no-dangling requirement for the region type of the current class, the no-dangling requirements for the fields' region types, and the class invariant of the parent class We do not require region parameters for primitive types, since primitive values can be copied and stored directly on the stack or they are part of an object. In order to keep the same notation, we use $prim\langle\rangle$ to denote a region annotated primitive type. Although null values are of object type, they are regarded as primitive values. The type of a null value is denoted by $\perp$.

The *region subtyping principle* allows an object from a region with longer lifetime to be assigned to a location where a region with a shorter lifetime is expected. This principle is illustrated by the subtyping rule [RegSub] of Fig. 3. This rule relies on the fact that once an object is allocated in a particular region, it stays within the same region and never migrates to another region. This property allows us to apply covariant subtyping to the region of the current object. However, the object fields are mutable (in general) and must therefore use invariant subtyping to ensure the soundness of subsumption. The other two rules, [SubClass] and [Null] from Fig. 3 denote the class subtyping and the fact that a null value can be assigned to any object, respectively.

$$\frac{}{\varphi{=}(x_1{\succeq}\hat{x}_1) \wedge \bigwedge_{i=2}^{n}(x_i{=}\hat{x}_i)} \; [\textbf{RegSub}]$$
$$\vdash cn\langle x_{1..n}\rangle{<:}cn\langle \hat{x}_{1..n}\rangle,\ \varphi$$

$$\frac{\textbf{class}\ cn\langle r_{1..n}\rangle\ \textbf{extends}\ cn'\langle r_{1..m}\rangle.. \in P' \quad n{\geq}m{\geq}p \quad \vdash cn'\langle x_{1..m}\rangle{<:}cn''\langle x'_{1..p}\rangle,\ \varphi}{\vdash cn\langle x_{1..n}\rangle{<:}cn''\langle x'_{1..p}\rangle,\ \varphi} \; [\textbf{SubClass}]$$

$$\frac{}{\vdash \perp{<:}cn\langle x_{1..n}\rangle,\ \mathit{true}} \; [\textbf{Null}]$$

Fig. 3. Region Subtyping Rules

Every method is decorated with zero or more region parameters; these parameters capture the regions used by each method's parameters (including `this`) and result. For simplicity, no other externally defined regions are made available for a method. Thus, all regions used in a method either are mapped to these region parameters or are localised by `letreg` in the method body. Each method also has a method precondition, φ expressed as a region lifetime constraint that is consistent with the operations performed in the method body. The method precondition also contains the class invariants of its parameters including the receiver and its result. The instance methods of a subclass can override the instance methods of the superclass.

Consider the `Pair` class in Fig. 4. As there are two fields in this class, a distinct region is introduced for each of them, `r2` for `fst` field and `r3` for `snd` field. The `Pair`

object is placed in the region r1. To ensure that every Pair instance satisfies the no-dangling requirement, the region lifetime constraint r2≥r1∧r3≥r1 is added to the class invariant.

```
class Pair⟨r1,r2,r3⟩ extends Object⟨r1⟩
   where r2≥r1 ∧ r3≥r1 {
   Object⟨r2⟩ fst;
   Object⟨r3⟩ snd;

void setSnd⟨r1,r2,r3,r4⟩(Object⟨r4⟩ o)
   where r4≥r3∧r2≥r1∧r3≥r1
    {snd=o;}
void swap⟨r1,r2,r3⟩() where r2=r3∧r2≥r1
   { Object⟨r2⟩ tmp=fst;fst=snd;snd=tmp}
Pair⟨r5,r6,r7⟩ exalloc⟨r1,r2,r3,r5,r6,r7⟩()
    where r7≥r5∧r6≥r5∧r2≥r1∧r3≥r1
  {letreg r in {
    Pair⟨r7,r7,r7⟩ p4;
    Pair⟨r,r,r⟩ p3;
    Pair⟨r5,r6,r7⟩ p2;
    Pair⟨r,r,r⟩ p1;
    p4 = new Pair⟨r7,r7,r7⟩(null,null);
    p3 = new Pair⟨r,r,r⟩(p4,null);
    p2 = new Pair⟨r5,r6,r7⟩(null,p4);
    p1 = new Pair⟨r,r,r⟩(p2,null);
    p1.setSnd⟨r,r,r,r⟩(p3); p2} }
}
```

Fig. 4. Region-Annotated Core-Java Program

Consider the setSnd, swap, and exalloc methods of the Pair class. A set of distinct region parameters are introduced for the methods' parameters, and the results, as shown in Fig. 4. The receiver regions are taken from the class definition. Moreover, the methods' region lifetime constraints are based on the possible operations of the respective methods. For example, due to an assignment operation and region subtyping, we have r4≥r3 for setSnd, while r2=r3 is present due to the swapping operation on the receiver object in the swap method. Though the swap method's region constraint is exclusively on the regions of the current object, we associate the constraint with the method. In this way, only those objects that might call the method are required to satisfy this constraint. The class invariants of methods' parameters (including the receiver and their result) are also added to the methods' region constraints. The exalloc method's body introduces a local region r using letreg. Since the p1 and p3 objects do not escape from the exalloc method's body, they are stored in the local region r. The p2 and p4 objects escape through the method result, therefore they are stored in the method result's regions r5 and r7, respectively.

3 Region Type System: Static Semantics

Our region type system guarantees that region-annotated Core-Java programs never create dangling references. To avoid variable name duplication, we assume that the local variables of the blocks and the arguments of the functions are uniquely renamed in a preprocessing phase. A part of region type checking rules are depicted in Fig. 5, with some auxiliary rules in Fig. 6 (a complete description of region type system is given in [11]). Judgments of the following forms are employed:

– $\vdash P$ denoting that a program P is well-typed.

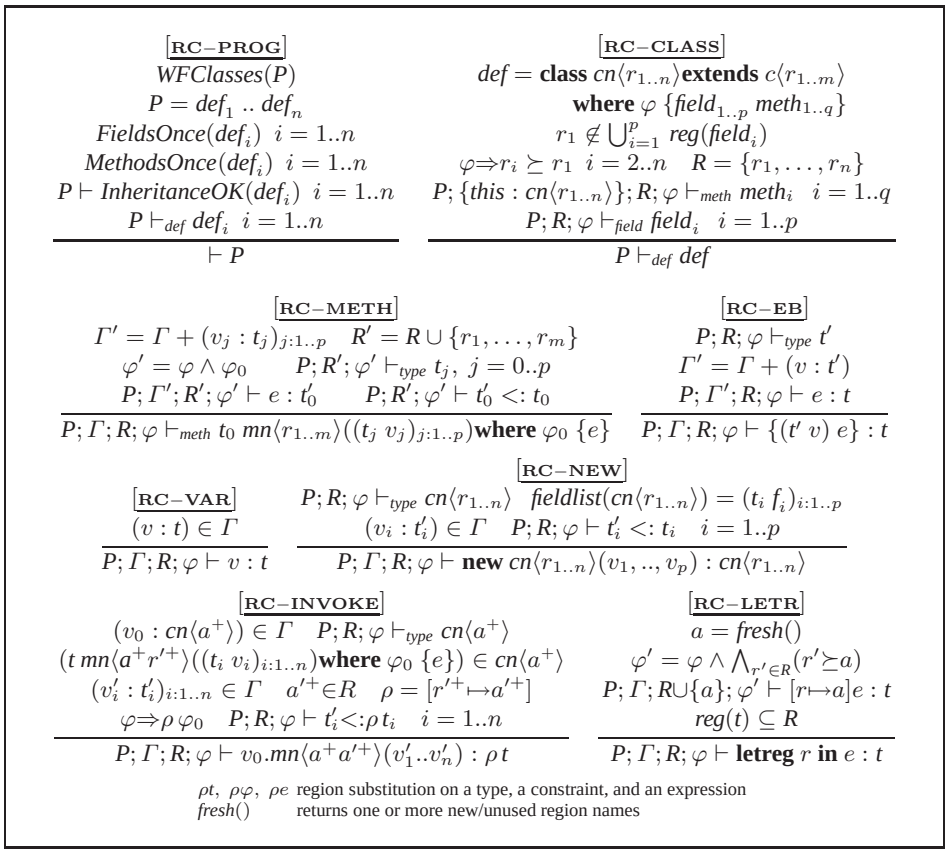

Fig. 5. Region Type Checking Rules

- $P \vdash_{def} def$ denoting that a class declaration *def* is well-formed.
- $P;\Gamma;R;\varphi \vdash_{meth} meth$ denoting that a method *meth* is well-defined with respect to the program *P*, the type environment Γ, the set of live regions *R*, and the region constraint φ.
- $P;\Gamma;R;\varphi \vdash e{:}t$ denoting that an expression e is well-typed with respect to the program *P*, the type environment Γ, the set of live regions *R*, and the region constraint φ.
- $P; R; \varphi \vdash_{type} t$ denoting that a type *t* is well-formed, namely, the regions of the type *t* are from the set of the live regions *R*, and the invariant of the type *t* is satisfied by the constraint context φ.
- $P; R \vdash_{constr} t,\ \varphi$ denoting that the regions of the type *t* are from the set of the live regions *R*, while φ is the invariant of the type *t*.
- $P; R; \varphi \vdash_{field} field$ denoting that the type of a field *field* is well-formed with respect to $\vdash_{type}$ judgment.
- $P; R; \varphi \vdash t{<:}t'$ denoting that the type *t* is a subtype of the type t', namely both types are well-formed and the region constraint of the subtyping relation (defined in Fig. 3) is satisfied by the constraint context φ.

The rule [RC−PROG] denotes that a region-annotated program is well-typed if all declared classes are well-typed. The predicates in the premise are used to capture the

standard well-formedness conditions for the object-oriented programs such as no duplicate definitions of classes and no cycle in the class hierarchy; no duplicate definitions of fields; no duplicate definitions of methods; and soundness of class subtyping and method overriding.

$$reg(\{\}) =_{def} \{\} \quad reg(\{v{:}\tau\langle r^*\rangle\} \cup \Gamma) =_{def} \{r^*\} \cup reg(\Gamma)$$

$$reg(\tau\langle r^*\rangle) =_{def} \{r^*\} \quad reg((\tau\langle r^*\rangle\ f)) =_{def} \{r^*\}$$

$$reg(r_1 \succeq r_2) =_{def} \{r_1, r_2\} \quad reg(r_1 = r_2) =_{def} \{r_1, r_2\}$$

$$reg(true) =_{def} \{\} \quad reg(\varphi_1 \wedge \varphi_2) =_{def} reg(\varphi_1) \cup reg(\varphi_2)$$

$$fieldlist(\textbf{Object}\langle r\rangle) =_{def} [\,]$$

$$\frac{\textbf{class}\ cn_1\langle r_{1..n}\rangle\ \textbf{extends}\ cn_2\langle r_{1..m}\rangle..\{(t_i\ f_i)_{i:1..p}..\} \in P' \quad \ell = fieldlist(\rho\ cn_2\langle r_{1..m}\rangle) \quad \rho = [r_i \mapsto x_i]_{i=1}^{n}}{fieldlist(cn_1\langle x_{1..n}\rangle) =_{def} \ell +\!\!+ [(\rho\ t_i)\ f_i]_{i=1}^{p}}$$

Fig. 6. Auxiliary Region Checking Rules

The rule [RC–CLASS] indicates that a class is well-formed if all its fields and methods are well-formed, and the class invariant ensures the necessary lifetime relations among class region parameters. In addition, the rule does not allow the first region of the class to be used by the region types of the fields. Using the first region on a field would break the object (region) subtyping (rule [RegSub] of Fig. 3). Function $reg(field_i)$ returns the region variables of a field type (see Fig. 6).

The rule [RC–METH] checks the well-formedness of a method declaration. Each region type is checked to be well-formed, that means its regions are in the current set of live regions and its invariant is satisfied by the current constraint context. The method body is checked using the type relation for expressions such that the gathered type has to be a subtype of the declared type.

Our type relation for expressions is defined in a syntax-directed fashion. Take note that region constraints of the variables are not checked at their uses ([RC–VAR]), but at their declaration sites ([RC–EB]). The region invariant of an object is also checked when that object is created ([RC–NEW]). In the rule for object creation ([RC–NEW]), the function $fieldlist(cn\langle x_{1..n}\rangle)$ returns a list comprising all declared and inherited fields of the class $cn\langle x_{1..n}\rangle$ and their region types according to the regions $x_1..x_n$ of the class cn (see Fig. 6). They are organized in an order determined by the constructor function.

The rule [RC–INVOKE] is used to check a method call. It ensures that the method region parameters are live regions and the method precondition is satisfied by the current constraint context as $\varphi \Rightarrow \rho\varphi_0$. A substitution ρ is computed for the method's formal region parameters. The current arguments are also checked to be subtypes of the method's formal parameters.

The rule [RC–LETR] is used to check a local region declaration. The local expression is checked with an extra live region a (that is a fresh region), and an extra constraint $\bigwedge_{r' \in R}(r' \succeq a)$ that ensures that newly introduced region is on the top of the region stack. The rule uses a region substitution on the expressions. Note that the region substitutions on expressions, constraints and types are defined as expected. The gathered region type of the local expression is checked to contain only live regions (from R excepting a). This guarantees that the localized region a does not escape. Function $reg(t)$ returns all region variables of t (see Fig. 6).

4 Dynamic Semantics

In this section we define the dynamic semantics of our region calculus. Our dynamic semantics rules use runtime checks to throw an error and to abort the execution, whenever the evaluation of a region-annotated Core-Java program tries to create a dangling reference. In Section 6 we prove that those runtime checks are redundant for well-typed programs, namely the evaluation of a well-typed region-annotated Core-Java program never creates a dangling reference. The dynamic semantics is defined as a small-step rewriting relation from machine states to machine states. A machine state is of the form $\langle \varpi, \Pi \rangle [e]$, where ϖ is the heap organized as a stack of regions, Π is the variable environment, and e is the current program. Our dynamic semantics was inspired by the previous work on abstract models of memory management [18] and region-based memory management [9, 13]. The following notations are used:

Region Variables :	$r, a \in$ *RegVar*
Offset :	$o \in$ *Offset*
Locations :	ℓ *or* $(r, o) \in$ *Location*=*RegVar*×*Offset*
Primitive Values :	$k \mid \texttt{null} \in$ *Prim*
Values :	$\delta \in$ *Value* = *Prim* ⊎ *Location*
Variable Environment :	$\Pi \in$ *VEnv* = *Var* $\rightharpoonup_{\text{fin}}$ *Value*
Field Environment :	$V \in$ *FEnv* = *FieldName* $\rightharpoonup_{\text{fin}}$ *Value*
Object Values :	$cn\langle r^* \rangle(V) \in$ *ObjVal* = *ClassName* × (*RegVar*)^n × *FEnv*
Store :	$\varpi \in$ *Store* = [ ] | [*r*↦*Rgn*]*Store*
Runtime Regions :	*Rgn* ∈ *Region* = *Offset* $\rightharpoonup_{\text{fin}}$ *ObjVal*

Regions are identified by region variables. We assume a denumerably infinite set of region variables, *RegVar*. The store ϖ is organized as a stack, that defines an ordered map from region variables, r to runtime regions *Rgn*. The notation $[r \mapsto Rgn]\varpi$ denotes a stack with the region r on the top, while [ ] denotes an empty store. The store can only be extended with new region variables. A runtime region *Rgn* is an unordered finite map from offsets to object values. We assume a denumerably infinite set of offsets, *Offset* for each runtime region *Rgn*.

The set of values that can be assigned to variables and fields is denoted by *Value*. Such a value is either a primitive value (a constant or a null value) or it is a location in the store. A location consists of a pair of a region variable and an offset.

An object value consists of a region type $cn\langle r^* \rangle$, and a field environment V mapping field names to values. V is not really an environment since it can only be updated, never extended. An update of field f with value δ is written as $V + \{f \mapsto \delta\}$.

The variable environment Π is a mapping *Var* $\rightharpoonup_{\text{fin}}$ *Value*, while the type environment Γ that corresponds to the runtime variable environment is also a mapping *Var* $\rightharpoonup_{\text{fin}}$ *Type*. To avoid variable name duplication, we assume that the local variables of the blocks and the arguments of the functions are uniquely renamed in a preprocessing phase.

Notation $f : A \rightharpoonup_{\text{fin}} B$ denotes a partial function from A to B with a finite domain, written $A = dom(f)$. We write $f + \{a \mapsto b\}$ for the function like f but mapping a to b (if $a \in dom(f)$ and $f(a) = c$ then $(f + \{a \mapsto b\})(a) = b$). The notation $\{\}$ (or $\emptyset$) stands for an undefined function. Given a function $f : A \rightharpoonup_{\text{fin}} B$, the notation $f - C$ denotes the function $f_1 : (A - C) \rightharpoonup_{\text{fin}} B$ such that $\forall x \in (A - C) \cdot f_1(x) = f(x)$.

We require some intermediate expressions for the small-step dynamic semantics to follow through. The intermediate expressions help our proof to use simpler induction techniques rather than a more elaborate co-induction machinery. The syntax of intermediate expressions is thus extended from the original expression syntax, as follows:

$$e ::= \ldots \mid (r, o) \mid \texttt{ret}(v, e) \mid \texttt{retr}(r, e)$$

The expression $\texttt{ret}(v, e)$ is used to capture the result of evaluating a local block, or the result of a method invocation. The variable associated with `ret` denotes either a block local variable or a method receiver or a method parameter. This variable is popped from the variable environment at the end of the block's evaluation. In the case of a method invocation there are multiple nested `ret`*s* which pop off the receiver and the method parameters from the variable environment at the end of the method's evaluation. The expression $\texttt{retr}(r, e)$ is used to pop off the top region, *r* of the store stack at the end of expression *e* evaluation.

Dynamic semantics rules of region annotated Core-Java are shown in Fig. 7 and Fig. 8. The evaluation judgment is of the form:

$$\langle \varpi, \Pi \rangle [e] \hookrightarrow \langle \varpi', \Pi' \rangle [e']$$

where ϖ (ϖ') denotes the store before (after) evaluation, while Π (Π') denotes the variable environment before (after) evaluation. The store ϖ organized as a stack establishes the outlive relations among regions at runtime. The function $ord(\varpi)$ returns the outlive relations for a given store. The function $dom(\varpi)$ returns the set of the store regions, while the function $location\_dom(\varpi)$ returns the set of all locations from the store. They are defined as follows:

$ord([r_1 \mapsto Rgn_1][r_2 \mapsto Rgn_2]\varpi) =_{def} (r_2 \succeq r_1) \wedge ord([r_2 \mapsto Rgn_2]\varpi)$
$ord([r \mapsto Rgn]) =_{def} true \quad ord([\,]) =_{def} true$
$dom([r \mapsto Rgn]\varpi) =_{def} \{r\} \cup dom(\varpi) \quad dom([r \mapsto \emptyset]\varpi) =_{def} \{r\} \cup dom(\varpi) \quad dom([\,]) =_{def} \emptyset$
$location\_dom(\varpi) =_{def} \{(r, o) \mid \varpi = \varpi_1[r \mapsto Rgn]\varpi_2 \wedge Rgn \neq \emptyset \wedge o \in dom(Rgn)\}$

Notation $\varpi(r)(o)$ denotes an access into the region r at the offset o, as follows:

$$\varpi(r)(o) =_{def} Rgn(o) \;\; where \;\; \varpi = \varpi_1[r \mapsto Rgn]\varpi_2$$

We define the meaning of *no-dangling references* property at runtime. The property refers to two kinds of references: (1) references from variable environment to store locations, and (2) references from store locations to other store locations. Note that the notion of *no-dangling references* was introduced in Fig. 1, and a reference is formalized as a location (r, o).

Definition 1. *(live location) A location* (r, o) *is* live *with respect to a store* ϖ*, if* $r \in dom(\varpi)$.

Definition 2. *(no-dangling)*

1. *A variable environment* Π *is* no-dangling *with respect to a store* ϖ *if for all* $v \in dom(\Pi)$,$\Pi(v)$ *is either a primitive value, or a live location* (r, o) *with respect to* ϖ.
2. *A runtime store* ϖ *is* no-dangling *if each region* $r_1 \in dom(\varpi)$ *contains only references to regions older than itself, that means that for each location* $(r_1, o) \in location\_dom(\varpi)$ *containing an object value* $\varpi(r_1)(o) = cn\langle r_{1..n} \rangle(V)$*, that object value satisfies the* non-dangling requirement for a class, *such that* $ord(\varpi) \Rightarrow \bigwedge_{i:2..n}(r_i \succeq r_1)$

$$\frac{v \in dom(\Pi)}{\langle \varpi, \Pi\rangle[v] \hookrightarrow \langle \varpi, \Pi\rangle[\Pi(v)]}\ [\text{D-VAR}] \qquad \frac{\Pi(v)=(r,o) \quad \varpi=\varpi_1[r\mapsto Rgn]\varpi_2 \quad Rgn(o)=cn\langle a^+\rangle(V)}{\langle \varpi, \Pi\rangle[v.f] \hookrightarrow \langle \varpi, \Pi\rangle[V(f)]}\ [\text{D-FD}]$$

$$\frac{lhs = v \mid v.f \quad \langle \varpi, \Pi\rangle[e] \hookrightarrow \langle \varpi', \Pi'\rangle[e']}{\langle \varpi, \Pi\rangle[lhs = e] \hookrightarrow \langle \varpi', \Pi'\rangle[lhs = e']}\ [\text{D-ASSGN1}] \qquad \frac{v \in dom(\Pi) \quad \Pi' = \Pi + \{v \mapsto \delta\} \quad \delta=(r_1, o_1) \ \wedge\ r_1 \in dom(\varpi)}{\langle \varpi, \Pi\rangle[v = \delta] \hookrightarrow \langle \varpi, \Pi'\rangle[()]}\ [\text{D-ASSGN2}]$$

$$\frac{v \in dom(\Pi) \quad \delta=(r_1, o_1) \ \wedge\ r_1 \notin dom(\varpi)}{\langle \varpi, \Pi\rangle[v=\delta] \hookrightarrow \texttt{danglingerr}}\ [\text{D-ASSGN2-DANGLERR}]$$

$$\frac{\Pi(v)=(a,o) \quad \varpi=\varpi_1[a\mapsto Rgn]\varpi_2 \quad Rgn(o)=cn\langle a^+\rangle(V) \quad Rgn' = Rgn + \{o \mapsto cn\langle a^+\rangle(V + \{f \mapsto \delta\})\} \quad \varpi' = \varpi_1[a \mapsto Rgn']\varpi_2 \quad \delta=(r_1,o_1) \ \wedge\ \textbf{ord}(\varpi) \Rightarrow (r_1 \succeq \textbf{fieldregion}(cn\langle a^+\rangle, f))}{\langle \varpi, \Pi\rangle[v.f = \delta] \hookrightarrow \langle \varpi', \Pi\rangle[()]}\ [\text{D-ASSGN3}]$$

$$\frac{\Pi(v)=(a,o) \quad \varpi=\varpi_1[a\mapsto Rgn]\varpi_2 \quad Rgn(o)=cn\langle a^+\rangle(V) \quad \delta=(r_1,o_1) \ \wedge\ \neg\,(\textbf{ord}(\varpi) \Rightarrow (r_1 \succeq \textbf{fieldregion}(cn\langle a^+\rangle, f)))}{\langle \varpi, \Pi\rangle[v.f = \delta] \hookrightarrow \texttt{danglingerr}}\ [\text{D-ASSGN3-DANGLERR}]$$

$$\frac{\begin{array}{c}\textbf{class}\ cn\langle r_{1..n}\rangle\ \textbf{extends}\ c\langle ...\rangle\ \textbf{where}\ \varphi_{inv}\ \{...\} \in P \qquad \textbf{ord}(\varpi) \Rightarrow \varphi_{inv} \\ \varpi=\varpi_1[r_1\mapsto Rgn]\varpi_2 \quad V=\{f_1\mapsto \Pi(v_1), ..., f_p \mapsto \Pi(v_p)\} \quad \textbf{fieldlist}(cn\langle r_{1..n}\rangle)=(t_i\ f_i)_{i:1..p} \\ \textit{if}\ \Pi(v_i)=(r_i', o_i')\ \textit{then}\ \textbf{ord}(\varpi) \Rightarrow (r_i' \succeq \textbf{fieldregion}(cn\langle r_{1..n}\rangle, f_i)) \quad i=1..p \\ o \notin dom(Rgn) \quad Rgn' = Rgn + \{o \mapsto cn\langle r_{1..n}\rangle(V)\} \quad \varpi' = \varpi_1[r_1 \mapsto Rgn']\varpi_2\end{array}}{\langle \varpi, \Pi\rangle[\textbf{new}\, cn\langle r_{1..n}\rangle(v_{1..p})] \hookrightarrow \langle \varpi', \Pi\rangle[(r_1, o)]}\ [\text{D-NEW}]$$

$$\frac{\begin{array}{c}\textbf{class}\ cn\langle r_{1..n}\rangle\ \textbf{extends}\ c\langle ...\rangle\ \textbf{where}\ \varphi_{inv}\ \{...\}\ \in\ P \\ V=\{f_1\mapsto \Pi(v_1), ..., f_p \mapsto \Pi(v_p)\} \quad \textbf{fieldlist}(cn\langle r_{1..n}\rangle)=(t_i\ f_i)_{i:1..p} \\ \neg(\textbf{ord}(\varpi) \Rightarrow \varphi_{inv}) \vee\ (\exists i \in \{1..p\} \cdot \Pi(v_i)=(r_i', o_i') \wedge \\ \qquad \neg(\textbf{ord}(\varpi) \Rightarrow (r_i' \succeq \textbf{fieldregion}(cn\langle r_{1..n}\rangle, f_i)))\end{array}}{\langle \varpi, \Pi\rangle[\textbf{new}\, cn\langle r_{1..n}\rangle(v_{1..p})] \hookrightarrow \texttt{danglingerr}}\ [\text{D-NEW-DANGLERR}]$$

$$\frac{\begin{array}{c}\{a^+, a'^+\} \subset \textbf{dom}(\varpi) \\ \Pi(v_0') = (a_1, o) \quad \varpi(a_1)(o) = cn\langle a^+\rangle(V) \\ (t_0\ mn\langle a^+ r'^+\rangle((t\,v)_{1..p})\textbf{where}\ \varphi\ \{e\}) \in cn\langle a^+\rangle \\ n_i = \textit{fresh}()\ \ i=0..p \quad \rho=[r'^+ \mapsto a'^+] \quad \Pi' = \Pi + \{n_i \mapsto \Pi(v_i')_{i:0..p}\} \\ e' = \texttt{ret}(n_0, \texttt{ret}(n_1, ..\texttt{ret}(n_p, [this \mapsto n_0][v_i \mapsto n_i]_{i:1}^p \rho e)))\end{array}}{\langle \varpi, \Pi\rangle[v_0'.mn\langle a^+ a'^+\rangle(v_{1..p}')] \hookrightarrow \langle \varpi, \Pi'\rangle[e']}\ [\text{D-INVOKE}]$$

$$\frac{\neg(r^+ \in \textbf{dom}(\varpi))}{\langle \varpi, \Pi\rangle[v.mn\langle r^+\rangle(v^*)] \hookrightarrow \texttt{danglingerr}}\ [\text{D-INVOKE-DANGLERR}]$$

Fig. 7. Dynamic Semantics for Region-Annotated Core-Java: Part I

$$\frac{n{=}\mathit{fresh}()\quad \Pi'{=}\Pi{+}\{(n{\mapsto}\mathbf{init}(t))\}\quad e'{=}\mathtt{ret}(n,e)}{\langle\varpi,\Pi\rangle[\{(t\,v)\ e\}]{\hookrightarrow}\langle\varpi,\Pi'\rangle[e']}\ [\textsc{D-EB}] \qquad \frac{\langle\varpi,\Pi\rangle[e]{\hookrightarrow}\langle\varpi',\Pi'\rangle[e']}{\langle\varpi,\Pi\rangle[\mathtt{ret}(v,e)]{\hookrightarrow}\langle\varpi',\Pi'\rangle[\mathtt{ret}(v,e')]}\ [\textsc{D-RET1}]$$

$$\frac{}{\langle\varpi,\Pi\rangle[\mathtt{ret}(v,\delta)]{\hookrightarrow}\langle\varpi,\Pi-\{v\}\rangle[\delta]}\ [\textsc{D-RET2}]$$

$$\frac{a{=}\mathit{fresh}()}{\langle\varpi,\Pi\rangle[\mathbf{letreg}\ r\ \mathbf{in}\ e]{\hookrightarrow}\langle[a{\mapsto}\emptyset]\varpi,\Pi\rangle[\mathtt{retr}(a,[r{\mapsto}a]e)]}\ [\textsc{D-LETR}]$$

$$\frac{\langle\varpi,\Pi\rangle[e]{\hookrightarrow}\langle\varpi',\Pi'\rangle[e']}{\langle\varpi,\Pi\rangle[\mathtt{retr}(a,e)]{\hookrightarrow}\langle\varpi',\Pi'\rangle[\mathtt{retr}(a,e')]}\ [\textsc{D-RETR1}]$$

$$\frac{\begin{array}{c}(\delta{=}(r,o)){\Rightarrow}(r{\in}\mathbf{dom}(\varpi))\\ \forall v{\in}\Pi\cdot(\Pi(v){=}(r,o)){\Rightarrow}(r{\in}\mathbf{dom}(\varpi))\\ \forall(r_1,o){\in}\mathit{location\_dom}(\varpi)\cdot(\varpi(r_1)(o){=}cn\langle r_{1..n}\rangle(V)){\Rightarrow}(r_{1..n}{\in}\mathbf{dom}(\varpi){\wedge}\\ \forall f\in dom(V)\ .\ V(f){=}(r_f,o_f)\wedge r_f{\in}\mathbf{dom}(\varpi))\end{array}}{\langle[a{\mapsto}\mathit{Rgn}]\varpi,\Pi\rangle[\mathtt{retr}(a,\delta)]{\hookrightarrow}\langle\varpi,\Pi\rangle[\delta]}\ [\textsc{D-RETR2}]$$

$$\frac{\begin{array}{c}\neg(a{=}a_1){\vee}\\ \neg((\delta{=}(r,o)){\Rightarrow}(r{\in}\mathbf{dom}(\varpi)))\vee\neg((\forall v{\in}\Pi\cdot(\Pi(v){=}(r,o)){\Rightarrow}(r{\in}\mathbf{dom}(\varpi))))\\ \vee\neg(\forall(r_1,o){\in}\mathit{location\_dom}(\varpi)\cdot(\varpi(r_1)(o){=}cn\langle r_{1..n}\rangle(V)){\Rightarrow}(r_{1..n}{\in}\mathbf{dom}(\varpi){\wedge}\\ \forall f\in dom(V)\ .\ V(f){=}(r_f,o_f)\wedge r_f{\in}\mathbf{dom}(\varpi)))\end{array}}{\langle[a{\mapsto}\mathit{Rgn}]\varpi,\Pi\rangle[\mathtt{retr}(a_1,\delta)]{\hookrightarrow}\mathtt{danglingerr}}\ [\textsc{D-RETR2-DANGLERR}]$$

$$\frac{\Pi(v){=}\mathit{true}}{\langle\varpi,\Pi\rangle[\mathbf{if}\ v\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2]{\hookrightarrow}\langle\varpi,\Pi\rangle[e_1]}\ [\textsc{D-IF1}] \qquad \frac{\Pi(v){=}\mathit{false}}{\langle\varpi,\Pi\rangle[\mathbf{if}\ v\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2]{\hookrightarrow}\langle\varpi,\Pi\rangle[e_2]}\ [\textsc{D-IF2}]$$

$$\frac{\Pi(v){=}\mathit{true}}{\langle\varpi,\Pi\rangle[\mathbf{while}\ v\ e]{\hookrightarrow}\langle\varpi,\Pi\rangle[e\,;\mathbf{while}\ v\ e]}\ [\textsc{D-LOOP1}] \qquad \frac{\Pi(v){=}\mathit{false}}{\langle\varpi,\Pi\rangle[\mathbf{while}\ v\ e]{\hookrightarrow}\langle\varpi,\Pi\rangle[()]}\ [\textsc{D-LOOP2}]$$

$$\frac{\langle\varpi,\Pi\rangle[e_1]{\hookrightarrow}\langle\varpi',\Pi'\rangle[e_1']}{\langle\varpi,\Pi\rangle[e_1\,;e_2]{\hookrightarrow}\langle\varpi',\Pi'\rangle[e_1'\,;e_2]}\ [\textsc{D-SEQ1}] \qquad \frac{}{\langle\varpi,\Pi\rangle[\delta_1\,;e_2]{\hookrightarrow}\langle\varpi,\Pi\rangle[e_2]}\ [\textsc{D-SEQ2}]$$

$$\frac{\Pi(v){=}\mathtt{null}}{\langle\varpi,\Pi\rangle[v.f]{\hookrightarrow}\mathtt{nullerr}}\ [\textsc{D-NULLERR1}] \qquad \frac{\Pi(v){=}\mathtt{null}}{\langle\varpi,\Pi\rangle[v.f=\delta]{\hookrightarrow}\mathtt{nullerr}}\ [\textsc{D-NULLERR2}] \qquad \frac{\Pi(v){=}\mathtt{null}}{\langle\varpi,\Pi\rangle[v.mn\langle a^*\rangle(u^*)]{\hookrightarrow}\mathtt{nullerr}}\ [\textsc{D-NULLERR3}]$$

Fig. 8. Dynamic Semantics for Region-Annotated Core-Java: Part II

and the current values of the fields are either primitives or references to regions older than those expected by the region type $cn\langle r_{1..n}\rangle$*, as follows:*

$\forall f \in dom(V)\ .\ V(f)=(r_f, o_f) \quad ord(\varpi)\Rightarrow r_f \succeq fieldregion(cn\langle r_{1..n}\rangle, f)$

Function $fieldregion(cn\langle r_{1..n}\rangle, f)$ *computes the region type of the class field f and then returns its first region where the field is expected to be stored.*

The dynamic semantics evaluation rules may yield two possible runtime errors, namely:

$$Error ::= \texttt{nullerr} \mid \texttt{danglingerr}$$

The first error `nullerr` is due to null pointers (by accessing fields or methods of null objects). The second error `danglingerr` is reported when a store updating operation or a variable environment updating operation creates a dangling reference. Our dynamic semantics rules use runtime checks to guarantee that a `danglingerr` error is reported (and the execution is aborted) whenever the program evaluation tries to create a dangling reference. There are five situations that require no-dangling reference checks at runtime:

- *Creation of a new object value.* Rule [D–NEW] checks whether the class invariant holds, $ord(\varpi)\Rightarrow\varphi_{inv}$ (mainly whether the fields regions $r_{i:2..n}$ outlive the region r_1 of the object). The initial value of a field is also checked to be stored in a region that outlives the expected region of that field $r'_i \succeq fieldregion(cn\langle r_{1..n}\rangle, f_i)$. The function $fieldlist(cn\langle r_{1..n}\rangle)$ is defined in Fig. 6.
- *Updating of an object's field.* Rule [D–ASSGN3] checks whether the region r_1 of the new location $\delta=(r_1, o_1)$ outlives the expected region for the object field f, $r_1 \succeq fieldregion(cn\langle a^+\rangle, f)$.
- *Updating a variable from the variable environment.* Rule [D–ASSGN2] checks whether the new location $\delta=(r_1, o_1)$ assigned to a variable is live, namely its region is in the current store, $r_1 \in dom(\varpi)$.
- *Deallocation of a region.* Rule [D–RETR2] checks whether the region *a* is on the top of the store stack. Then it checks whether a reference to *a* does not escape neither through the value result δ, nor through the program variable environment Π, nor through the object values of the store ϖ. Note that when a new region is allocated, in rule [D–LETR], a fresh region name is used in order to avoid region name duplication in the store.
- *Calling a method.* Rule [D–INVOKE] checks whether the method's region arguments are in the current store and then prepares the variable environment for the method's body execution.

The corresponding rules [D–NEW–DANGLERR], [D–ASSGN3–DANGLERR], [D–ASSGN2–DANGLERR], [D–RETR2–DANGLERR], and [D–INVOKE–DANGLERR] generate a `danglingerr` error due to the failure of their runtime checks. In the rules [D–ASSGN2], [D–ASSGN3], and [D–LOOP2] the result **()** denotes the singleton value of type **void**. Note that the type **void** is assumed to be isomorphic to type **unit**. In rule [D–EB], the locally declared variable is assigned, with the help of the function **init**, an initial value according to its type as follows:

$$\begin{array}{lll} \textbf{init}(t) =_{def} & case\ t\ of & \\ \quad boolean & \rightarrow & false \\ \quad int & \rightarrow & 0 \\ \quad cn\langle r_{1..n}\rangle & \rightarrow & \texttt{null} \end{array}$$

5 Extended Static Semantics

In this section we extend our static semantics rules from Section 3 to include the new intermediate constructions introduced by the small-step dynamic semantics rules in Section 4.

First we define a *valid program* using a novel syntactic condition *valid(e)*, that restricts the places where the intermediate constructions may occur in a program.

Definition 3. *(valid program)*

1. *A program is a* valid program *if all the program's classes are valid classes.*
2. *A class is a* valid class *if all the class's methods are valid methods.*
3. *A method is a* valid method *if the method's body e is a valid block expression such that retvars(e)=$\emptyset$ and retregs(e)=$\emptyset$.*
4. *Expression e is a* valid expression *if the predicate valid(e) holds, where valid(e) is defined as follows:*

$$
\begin{array}{lll}
\textit{valid}(e) =_{def} \textit{case e of} & & \\
\quad \{(t\ v)\ e\} & \rightarrow & \textit{retvars}(e)=\emptyset \wedge \textit{retregs}(e)=\emptyset \\
\quad \textit{lhs} = e & \rightarrow & \textit{retvars}(e) \cap \textit{vars}(\textit{lhs})=\emptyset \wedge \textit{valid}(e) \\
\quad e_1 \textbf{ ; } e_2 & \rightarrow & \textit{retregs}(e_2)=\emptyset \wedge \textit{retvars}(e_2)=\emptyset \wedge \textit{valid}(e_1) \\
 & & \wedge \textit{retvars}(e_1) \cap \textit{vars}(e_2)=\emptyset \wedge \textit{retregs}(e_1) \cap \textit{regs}(e_2)=\emptyset \\
\quad \textbf{if } v \textbf{ then } e_1 \textbf{ else } e_2 & \rightarrow & \textit{retregs}(e_1)=\emptyset \wedge \textit{retvars}(e_1)=\emptyset \\
 & & \wedge \textit{retregs}(e_2)=\emptyset \wedge \textit{retvars}(e_2)=\emptyset \\
\quad \textbf{while } v\ e \mid \textbf{letreg } r \textbf{ in } e & \rightarrow & \textit{retregs}(e)=\emptyset \wedge \textit{retvars}(e)=\emptyset \\
\quad \texttt{ret}(v, e) & \rightarrow & v \notin \textit{retvars}(e) \wedge \textit{valid}(e) \\
\quad \texttt{retr}(r, e) & \rightarrow & r \notin \textit{retregs}(e) \wedge \textit{valid}(e) \\
\quad \textit{otherwise} & \rightarrow & \textit{true}
\end{array}
$$

This condition does not restrict source-level region calculus, since intermediate constructions are generated during the program evaluation. A source language Core-Java program is by default a valid program since it does not contain any intermediate expression. The above condition is based on the functions *vars*(e), *retvars*(e), *regs*(e), and *retregs*(e) which are defined as follows:

Definition 4. *1. The function vars(e) computes the set of all program variables which occur in the expression e, excepting those variables introduced by e's block subexpressions, as follows:*

$$
\begin{array}{lll}
\textit{vars}(e) =_{def} \textit{case e of} & & \\
\quad \texttt{ret}(v, e) & \rightarrow & \{v\} \cup \textit{vars}(e) \\
\quad \{(t\ v)\ \ e\} & \rightarrow & \textit{vars}(e) \setminus \{v\} \\
\quad \texttt{retr}(r, e) \mid \textbf{letreg } r \textbf{ in } e & \rightarrow & \textit{vars}(e) \\
\quad v.f = e \mid v = e \mid \textbf{while } v\ e & \rightarrow & \{v\} \cup \textit{vars}(e) \\
\quad v.f \mid v & \rightarrow & \{v\} \\
\quad \textbf{if } v \textbf{ then } e_1 \textbf{ else } e_2 & \rightarrow & \{v\} \cup \textit{vars}(e_1) \cup \textit{vars}(e_2) \\
\quad e_1 \textbf{ ; } e_2 & \rightarrow & \textit{vars}(e_1) \cup \textit{vars}(e_2) \\
\quad \textbf{new } cn\langle r^+ \rangle (v^*) & \rightarrow & \{v^*\} \\
\quad v.mn\langle v^* \rangle (v^*) & \rightarrow & \{v\} \cup \{v^*\} \\
\quad \textit{otherwise} & \rightarrow & \emptyset
\end{array}
$$

2. *The function* $retvars(e)$ *computes the set of all program variables which occur in the* ret *subexpressions of the expression* e*, as follows:*

$$
\begin{array}{lll}
retvars(e) =_{def} \textit{ case } e \textit{ of} & & \\
\quad \texttt{ret}(v, e) & \rightarrow & \{v\} \cup retvars(e) \\
\quad \texttt{retr}(r, e) \mid v.f = e \mid v = e \mid \{(t\ v)\ e\} & \rightarrow & retvars(e) \\
\quad \textbf{while } v\ e \mid \textbf{letreg } r \textbf{ in } e & \rightarrow & retvars(e) \\
\quad e_1 \textbf{ ; } e_2 \mid \textbf{if } v \textbf{ then } e_1 \textbf{ else } e_2 & \rightarrow & retvars(e_1) \cup retvars(e_2) \\
\quad otherwise & \rightarrow & \emptyset
\end{array}
$$

3. *The function* $regs(e)$ *computes the set of all region variables which occur in the expression* e*, excepting those regions introduced by* e*'s letreg subexpressions, as follows:*

$$
\begin{array}{lll}
regs(e) =_{def} \textit{ case } e \textit{ of} & & \\
\quad \{(t\ v)\ \ e\} & \rightarrow & reg(t) \cup regs(e) \\
\quad \texttt{retr}(r, e) & \rightarrow & \{r\} \cup regs(e) \\
\quad \textbf{letreg } r \textbf{ in } e & \rightarrow & regs(e) \setminus \{r\} \\
\quad \texttt{ret}(v, e) \mid v.f = e \mid v = e \mid \textbf{while } v\ e & \rightarrow & regs(e) \\
\quad (r, o) & \rightarrow & \{r\} \\
\quad \textbf{if } v \textbf{ then } e_1 \textbf{ else } e_2 \mid e_1 \textbf{ ; } e_2 & \rightarrow & regs(e_1) \cup regs(e_2) \\
\quad \textbf{new } cn\langle r^+\rangle\textbf{(}v^*\textbf{)} \mid v.mn\langle r^+\rangle\textbf{(}v^*\textbf{)} & \rightarrow & \{r^+\} \\
\quad otherwise & \rightarrow & \emptyset
\end{array}
$$

where reg(t) is defined in the Figure 6.

4. *The function* $retregs(e)$ *computes the set of all region variables which occur in the* retr *subexpressions of the expression* e*, as follows:*

$$
\begin{array}{lll}
retregs(e) =_{def} \textit{ case } e \textit{ of} & & \\
\quad \texttt{retr}(r, e) & \rightarrow & \{r\} \cup\ retregs(e) \\
\quad \texttt{ret}(v, e) \mid v.f = e \mid v = e \mid \{(t\ v)\ e\} & \rightarrow & retregs(e) \\
\quad \textbf{while } v\ e \mid \textbf{letreg } r \textbf{ in } e & \rightarrow & retregs(e) \\
\quad e_1 \textbf{ ; } e_2 \mid \textbf{if } v \textbf{ then } e_1 \textbf{ else } e_2 & \rightarrow & retregs(e_1) \cup retregs(e_2) \\
\quad otherwise & \rightarrow & \emptyset
\end{array}
$$

In order to describe the type of each location, we introduce a *store typing*. This ensures that objects created in the store during run-time are type-wise consistent with those captured by the static semantics. Store typing is conventionally used to link static and dynamic semantics [20]. In our case, it is denoted by Σ, as follows:

$$\Sigma \in \mathit{StoreType} = \mathit{RegVar} \rightharpoonup_{\text{fin}} \mathit{Offset} \rightharpoonup_{\text{fin}} \mathit{Type}$$

The judgments of static semantics are extended with store typing, as follows:

$$P; \Gamma; R; \varphi; \Sigma \vdash e : t$$

For a store typing $\Sigma : R \rightharpoonup_{\text{fin}} O \rightharpoonup_{\text{fin}} \mathit{Type}$, a region r, a location (r, o), and a type t we also introduce the following notations:

$$
\begin{array}{l}
dom(\Sigma) = R \qquad \Sigma(r)(o) = f(o),\ where\ f = \Sigma(r) \\
location\_dom(\Sigma) =_{def} \{(r, o) \mid r \in dom(\Sigma) \wedge f = \Sigma(r) \wedge f \neq \emptyset \wedge o \in dom(f)\} \\
\Sigma - r =_{def} \Sigma_1\ such\ that\ \Sigma_1 : (R - \{r\}) \rightharpoonup_{\text{fin}} O \rightharpoonup_{\text{fin}} \mathit{Type} \wedge \forall r' \in (R - r) \cdot \Sigma_1(r') = \Sigma(r') \\
\Sigma + r =_{def} \Sigma_2\ such\ that\ \Sigma_2 : (R \cup \{r\}) \rightharpoonup_{\text{fin}} O \rightharpoonup_{\text{fin}} \mathit{Type} \wedge \Sigma_2(r) = \emptyset \wedge \forall r' \in R \cdot \Sigma_2(r') = \Sigma(r') \\
\Sigma - (r, o) =_{def} \Sigma_3\ such\ that\ \Sigma_3 : R \rightharpoonup_{\text{fin}} O \rightharpoonup_{\text{fin}} \mathit{Type} \\
\qquad \wedge r \in R \wedge \Sigma_3(r) = \Sigma(r) - \{o\} \wedge \forall r' \in (R - r) \cdot \Sigma_3(r') = \Sigma(r') \\
\Sigma + ((r, o) : t) =_{def} \Sigma_4\ such\ that\ \Sigma_4 : R \rightharpoonup_{\text{fin}} O \rightharpoonup_{\text{fin}} \mathit{Type} \\
\qquad \wedge r \in R \wedge \Sigma_4(r) = \Sigma(r) + \{o \mapsto t\} \wedge \forall r' \in (R - r) \cdot \Sigma_4(r') = \Sigma(r')
\end{array}
$$

$$\frac{}{P;\Gamma;R;\varphi;\Sigma \vdash (r,o) : t}\ [\textsc{rc-location}]\ \ r\in R\ \ \Sigma(r)(o) = t$$

$$\frac{P;R;\varphi \vdash_{type} cn\langle r_{1..n}\rangle\ \ fieldlist(cn\langle r_{1..n}\rangle) = (t_i\ f_i)_{i:1..p} \quad P;\Gamma;R;\varphi;\Sigma \vdash V(f_i) : t_i' \quad P;R;\varphi \vdash t_i' <: t_i \quad i{=}1..p}{P;\Gamma;R;\varphi;\Sigma \vdash cn\langle r_{1..n}\rangle(V) : cn\langle r_{1..n}\rangle}\ [\textsc{rc-ObjVal}]$$

$$\frac{v\in\Gamma \quad P;\Gamma;R;\varphi;\Sigma \vdash e : t}{P;\Gamma;R;\varphi;\Sigma \vdash \texttt{ret}(v,e) : t}\ [\textsc{rc-ret}] \qquad \frac{P;\Gamma;R;\varphi;\Sigma \vdash e : t' \quad P;R;\varphi \vdash t' <: t}{P;\Gamma;R;\varphi;\Sigma \vdash e : t}\ [\textsc{subsumption}]$$

$$\frac{\begin{array}{c}a\in R \quad R_t{=}R{-}lreg(e){-}\{a\} \quad \varphi \Rightarrow \bigwedge_{r\in R_t}(r \succeq a) \\ reg(t)\subseteq R_t \quad reg(\Gamma{-}lvar(e)) \subseteq R_t \quad P;\Gamma;R;\varphi;\Sigma \vdash e : t\end{array}}{P;\Gamma;R;\varphi;\Sigma \vdash \texttt{retr}(a,e) : t}\ [\textsc{rc-retr}]$$

Fig. 9. Region Type Checking Rules for Valid Intermediate Expressions

The judgments of the new intermediate expressions are presented in Fig. 9. They assume that the expressions are valid with respect to the Definition 3. The first two rules [RC–LOCATION] and [RC–ObjVal] are used to type the store, either a location or an object value (i.e. a location's content). Rule [RC–ObjVal] preserves the same invariants as those of the rule [RC–NEW]. Rule [RC–RET] ensures that the variable to be popped off, *v* is in the current environment Γ. The subsumption rule [SUBSUMPTION] simplifies the next theorems and their proofs.

Rule [RC–RETR] is similar to rule [RC–LETR], but it takes into account the evaluation of the expression $\texttt{retr}(r,e)$. The first check ensures that the region to be deallocated, *a* is in R. The R_t denotes the regions from R which are different than *a* and are not younger than *a*. Note that *lreg*(e) denotes the regions which are younger than *a*. The second check ensures that our type system uses only lexically scoped regions such that the region to be deallocated, *a* is always on the top of the regions stack. The third and the fourth check ensure that the region *a* and the regions younger than *a* do not escape either through the result or through the live variables of the type environment. Note that *lvar*(e) denotes the local variables of the expression e which are deallocated from the variable environment during the evaluation of e.

The rules from Fig. 9 are using the functions *lvar*(e), *lreg*(e), and *lloc*(e) which are defined as follows:

Definition 5. *Using the evaluation rules from Fig. 7 and Fig. 8*

1. *The function lvar*(e) *estimates the set of variables which may be popped off from the variable environment* Π *during the evaluation of the valid expression* e *(note that only* $\texttt{ret}(v,e)$ *may affect* Π*), as follows:*

$$\begin{array}{lll} lvar(e) =_{def} \textit{case e of} & & \\ \quad \texttt{ret}(v,e) & \rightarrow & \{v\} \cup\ lvar(e) \\ \quad \texttt{retr}(r,e) \mid lhs = e \mid e\texttt{;}e_1 & \rightarrow & lvar(e) \\ \quad otherwise & \rightarrow & \emptyset \end{array}$$

2. *The function $lreg(e)$ estimates the set of regions which may be popped off from the store ϖ during the evaluation of the valid expression e (note that only* `retr`(r,e) *may affect ϖ), as follows:*

$$\begin{array}{lll} lreg(e) =_{def} \textit{ case } e \textit{ of} & & \\ \quad \texttt{retr}(r,e) & \rightarrow & \{r\} \cup lreg(e) \\ \quad \texttt{ret}(v,e) \mid lhs = e \mid e\,;e_1 & \rightarrow & lreg(e) \\ \quad otherwise & \rightarrow & \emptyset \end{array}$$

3. *The function $lloc(e)$ estimates the new location which may be created into an existing region during one evaluation step of the valid expression e (note that only* **new** *may create a new location), as follows:*

$$\begin{array}{lll} lloc(e) =_{def} \textit{ case } e \textit{ of} & & \\ \quad \textbf{new}\ cn\langle r_1,..,r_n\rangle(v^*) & \rightarrow & \{(r_1,o)\} \\ \quad \texttt{ret}(v,e) \mid \texttt{retr}(r,e) \mid lhs = e \mid e\,;e_1 & \rightarrow & lloc(e) \\ \quad otherwise & \rightarrow & \emptyset \end{array}$$

where the offset o of the region r is the offset where the next allocation in r is done.

6 Soundness Theorems

In this section we prove the soundness of our region calculus, namely that a valid program well-typed by our type system never creates dangling references. We use a syntactic proof method [25], based on a subject reduction theorem and a progress theorem.

First we define the consistency relationship between the static and dynamic semantics, namely a relationship between what we can estimate at compile-time and what can happen during run-time execution.

Definition 6. *(consistency relationship)*

A run-time environment (ϖ, Π) is consistent *with a static environment $(\Gamma, R, \varphi, \Sigma)$, written $\Gamma, R, \varphi, \Sigma \vDash \langle \varpi, \Pi \rangle$, if the following judgment holds:*

$$dom(\Gamma){=}dom(\Pi) \qquad \forall v \in dom(\Pi) \cdot P; \Gamma; R; \varphi; \Sigma \vdash \Pi(v) : \Gamma(v) \qquad reg(\Gamma){\subseteq}R$$
$$location\_dom(\Sigma){=}location\_dom(\varpi) \qquad dom(\Sigma){=}dom(\varpi) \qquad R{=}dom(\varpi)$$
$$ord(\varpi){\Rightarrow}\varphi \qquad \forall (r,o){\in}location\_dom(\varpi) \cdot P; \Gamma; R; \varphi; \Sigma \vdash \varpi(r)(o) : \Sigma(r)(o)$$

Note that $\varpi(r)(o)$ returns an object value $cn\langle r^*\rangle(V)$ whose type is $cn\langle r^*\rangle$. In our instrumented operational semantics an object value and its type are stored together.

The subject reduction theorem ensures that the region type is preserved during the execution of a valid program, as follows:

Theorem 1. *(Subject Reduction): If*

$$valid(e) \quad P; \Gamma; R; \varphi; \Sigma \vdash e : t$$
$$\Gamma, R, \varphi, \Sigma \vDash \langle \varpi, \Pi \rangle$$
$$\langle \varpi, \Pi \rangle[e] \hookrightarrow \langle \varpi', \Pi' \rangle[e']$$

then there exist Σ', Γ', R', and φ', such that

$$(\Sigma' - (lreg(e') - lreg(e))) - (lloc(e) - lloc(e')) = \Sigma - (lreg(e) - lreg(e'))$$
$$\Gamma' - (lvar(e') - lvar(e)) = \Gamma - (lvar(e) - lvar(e'))$$
$$R' - (lreg(e') - lreg(e)) = R - (lreg(e) - lreg(e'))$$
$$\varphi' - (lreg(e') - lreg(e)) \Rightarrow \varphi - (lreg(e) - lreg(e'))$$
$$\Gamma', R', \varphi', \Sigma' \vDash \langle \varpi', \Pi' \rangle$$
$$valid(e') \quad P; \Gamma'; R'; \varphi'; \Sigma' \vdash e' : t.$$

Proof: By structural induction on e. The detailed proof is in [11].

Although the hypothesis of the above theorem contains an evaluation relation, the proof does not use the run-time checks associated with the evaluation rules to prove that the result of the evaluation (result and dynamic environment) is well-typed, valid and consistent.

The progress theorem guarantees that the execution of a valid program cannot generate `danglingerr` errors, by proving that those run-time checks are redundant for a well-typed valid program (the run-time checks are proved by the static semantics).

Theorem 2. *(Progress) If*

$$\begin{array}{c} \mathit{valid}(e) \quad P; \Gamma; R; \varphi; \Sigma \vdash e : t \\ \Gamma, R, \varphi, \Sigma \vDash \langle \varpi, \Pi \rangle \end{array}$$

then either

- *e is a value, or*
- $\langle \varpi, \Pi \rangle[e] \hookrightarrow$ `nullerr` *or*
- *there exist* ϖ', Π', e' *such that* $\langle \varpi, \Pi \rangle[e] \hookrightarrow \langle \varpi', \Pi' \rangle[e']$.

Proof: By induction over the depth of the type derivation for expression e. The detailed proof is in [11].

We conclude with the following soundness theorem for region annotated Core-Java. The theorem states that if a valid program is well-typed and is evaluated in a run-time environment consistent with the static environment, the result of a finite number of reduction steps (denoted by $\hookrightarrow^*$) is (1) either an error different from a dangling error, (2) or a value, (3) or that the program diverges (namely after a finite number of reduction steps there still exists one more reduction step). The evaluation never reports dangling errors, namely the program never creates dangling references.

Theorem 3. *(Soundness) Given a well-typed valid Core-Java program* $P{=}def^*$ *and the main function* $(void\ \mathbf{main}(void)\{e_0\}) \in P$*, where* e_0 *is a well-typed valid closed term (without free regions and free variables), such that* $retvars(e_0){=}\emptyset \wedge retregs(e_0){=}\emptyset$ *and* $P; \Gamma_0; R_0; \varphi_0; \Sigma_0 \vdash e_0 : void$*, where* $\Gamma_0{=}\emptyset$*,* $R_0{=}\emptyset$*,* $\varphi_0{=}true$*, and* $\Sigma_0{=}\emptyset$ *. Starting from the initial run-time environment* $\langle \varpi_0, \Pi_0 \rangle$*, where* $\varpi_0{=}[\,]$*,* $\Pi_0{=}\emptyset$*, such that* $\Gamma_0, R_0, \varphi_0, \Sigma_0 \vDash \langle \varpi_0, \Pi_0 \rangle$*. Then either*

$$(1) \qquad \langle \varpi_0, \Pi_0 \rangle[e_0] \hookrightarrow^* \texttt{nullerr}$$

or there exist a store ϖ*, a variable environment* Π*, a value* δ*, a type environment* Γ*, a set of regions* R*, a region constraint* φ*, a store typing* Σ *such that*

$$(2) \qquad \langle \varpi_0, \Pi_0 \rangle[e_0] \hookrightarrow^* \langle \varpi, \Pi \rangle[\delta] \quad \Gamma, R, \varphi, \Sigma \vDash \langle \varpi, \Pi \rangle \quad P; \Gamma; R; \varphi; \Sigma \vdash \delta : void$$

or for a store ϖ*, a variable environment* Π*, a valid expression* e*, a type environment* Γ*, a set of regions* R*, a region constraint* φ*, a store typing* Σ *such that*

$$\langle \varpi_0, \Pi_0 \rangle[e_0] \hookrightarrow^* \langle \varpi, \Pi \rangle[e] \quad \Gamma, R, \varphi, \Sigma \vDash \langle \varpi, \Pi \rangle \quad P; \Gamma; R; \varphi; \Sigma \vdash e : void \quad \mathit{valid}(e)$$

there exist a store ϖ', a variable environment Π', an expression e', a type environment Γ', a set of regions R', a region constraint φ', a store typing Σ' such that

(3) $\langle \varpi, \Pi \rangle[e] \hookrightarrow \langle \varpi', \Pi' \rangle[e']$ $\Gamma', R', \varphi', \Sigma' \vDash \langle \varpi', \Pi' \rangle$ $P;\Gamma';R';\varphi';\Sigma' \vdash e' : void$ $valid(e')$

Proof: The proof is an induction on the number of the reduction steps. We can repeatedly use the progress theorem (Theorem 2) to prove that there is a reduction step and then the preservation theorem (Theorem 1) to prove that the run-time environment after evaluation is still well-typed and the evaluation result is valid.

7 Conclusion

We have considered a region calculus consisting of an object-oriented core language annotated with regions. We have defined the dynamic semantics for our region calculus based on a simpler small-step rewriting relation. Some of the region calculus constructions (e.g. `letreg`) are firstly evaluated to intermediate constructions. Therefore the static semantics must also be extended to include these new intermediate constructions. We have used a novel syntactic condition (*valid(e)*) to restrict the places where the intermediate constructions may occur in a program. This condition does not restrict source-level region calculus, since intermediate constructions are generated during the program evaluation. Our dynamic semantics is instrumented with runtime checks to guarantee that a special `danglingerr` error is reported whenever the program evaluation tries to create a dangling reference. We have defined an important consistency relationship between the static and dynamic semantics. A store typing technique is used to ensure that objects created in the store during run-time are type-wise consistent with those captured by the static semantics. We have proven the soundness of the region calculus by using a syntactic proof method [25], based on subject reduction and progress. The subject reduction theorem ensures that the region type of a valid program is preserved during the evaluation. The progress theorem guarantees that the evaluation of a valid program cannot generate `danglingerr` errors (namely those runtime checks are redundant for a well-typed valid program). We have proven both theorems in a modular fashion using just a simple induction. This simple soundness proof adds confidence to our region-based memory inference and execution systems.

References

1. David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 285–298, 2003.
2. L. Birkedal and M. Tofte. A constraint-based region inference algorithm. *Theoretical Computer Science*, 258(1–2):299–392, 2001.
3. G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.
4. Gerard Boudol. Typing safe deallocation. In *European Symposium on Programming (ESOP)*, pages 116–130, 2008.

5. C. Boyapati, A. Salcianu, W. Beebee, and M. Rinard. Ownership Types for Safe Region-Based Memory Management in Real-Time Java. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 324–337, 2003.
6. C. Calcagno. Stratified operational semantics for safety and correctness of the region calculus. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 155–165, 2001.
7. C. Calcagno, S. Helsen, and P. Thiemann. Syntactic type soundness results for the region calculus. *Information and Computation*, 173(2):199–221, 2002.
8. Wei-Ngan Chin, Florin Craciun, Shengchao Qin, and Martin C. Rinard. Region inference for an object-oriented language. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 243–254, 2004.
9. M. V. Christiansen and P. Velschow. Region-Based Memory Management in Java. Master's Thesis, Department of Computer Science (DIKU), University of Copenhagen, 1998.
10. Florin Craciun, Hong Yaw Goh, and Wei-Ngan Chin. A framework for object-oriented program analyses via Core-Java. In *IEEE International Conference on Intelligent Computer Communication and Processing (ICCP)*, pages 197–205, Cluj-Napoca, Romania, 2006.
11. Florin Craciun, Shengchao Qin, and Wei-Ngan Chin. A Formal Soundness Proof of Region-based Memory Management for Object-Oriented Paradigm. Technical report, Department of Computer Science, Durham University, UK., April 2008. Available at http://www.durham.ac.uk/shengchao.qin/papers/reg_cal_proof.pdf.
12. Martin Elsman. Garbage collection safety for region-based memory management. In *ACM Workshop on Types in Language Design and Implementation (TLDI)*, pages 123–134, 2003.
13. D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-Based Memory Management in Cyclone. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 282–293, 2002.
14. S. Helsen. *Region-Based Program Specialization*. PhD thesis, Universität Freiburg, 2002.
15. Simon Helsen and Peter Thiemann. Syntactic type soundness for the region calculus. *Electronic Notes in Theoretical Computer Science*, 41(3), 2000.
16. A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 132–146, 1999.
17. Gregory Morrisett. *Compiling with Types*. PhD thesis, Carnegie Mellon University, 1995.
18. J. Gregory Morrisett, Matthias Felleisen, and Robert Harper. Abstract Models of Memory Management. In *ACM Conference Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pages 66–77, 1995.
19. H. Niss. *Regions are imperative. Unscoped regions and control-sensitive memory management*. PhD thesis, University of Copenhagen, 2002.
20. B. Pierce. *Types and Programming Languages*. The MIT Press, 2002.
21. M. Tofte and L. Birkedal. A region inference algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(4):734–767, 1998.
22. M. Tofte and J. Talpin. Implementing the Call-By-Value λ-calculus Using a Stack of Regions. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 188–201, 1994.
23. M. Tofte and J. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.
24. Paul R. Wilson. Uniprocessor garbage collection techniques. In *International Workshop on Memory Management (IWMM)*, pages 1–42, 1992.
25. Andrew K. Wright and Matthias Felleisen. A Syntactic Approach to Type Soundness. *Information Computation*, 115(1):38–94, 1994.